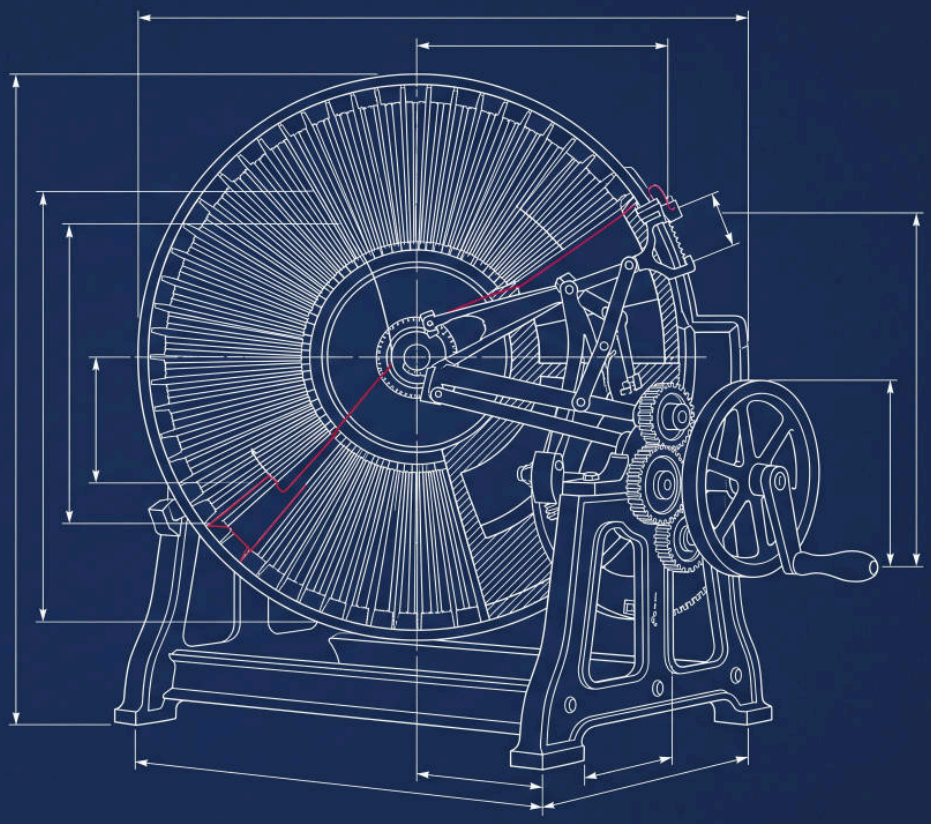

CHAPTER SIX

Chat with PDF End to End

SECTIONS 6.1 TO 6.11

Retrieval augmented generation is the most built and least engineered pattern in the field. The demo takes an afternoon. The version that survives documents you did not choose takes a quarter.

CHAPTER SIX

Contents

PART II · RETRIEVAL

Retrieval augmented generation is the most built and least engineered pattern in the field. The demo takes an afternoon. The version that survives contact with documents you did not choose takes a quarter, and almost all of the difference is in ingestion.

6.1 What RAG Is Actually Solving

The naive framing is that RAG gives the model knowledge it lacks. That framing produces bad architectures, because it suggests the goal is to get relevant text in front of the model and stops there.

The precise framing, using the physics from Chapter 2: **RAG is a context construction problem operating under a hard set of constraints.** You have a corpus far larger than any context window, a query whose information need is under-specified, and a model whose accuracy degrades with distractors and with position. The job is to select *the smallest set of passages that contains the answer*, in a form the model can use, with enough provenance that the answer can be cited and checked.

Every clause in that sentence is load bearing, and the tutorial pipeline addresses only the first.

The constraint	Where it came from	What it forces
The window is a budget, not a container	2.10	Named regions with ceilings, not concatenation
Attention degrades positionally	2.7	Evidence order is engineered, not incidental
Distractors dilute even when the answer is present	2.10	Precision beats recall in the assembled prompt
Cosine similarity is topical, not answer-bearing	2.9	Reranking is structural (Ch 7)
Approximate indexes lose recall silently	2.9	Index recall must be measured
Fluency is uncorrelated with correctness	2.1	Grounding verification and an abstain path (6.9)

The reframing that changes what you build. Stop asking "how do I get the documents to the model" and start asking "what is the minimum evidence that answers this, and can I prove the answer came from it". The first question produces a pipeline. The second produces an architecture, and it is the one that can be trusted by someone whose job depends on the answer.

There is a prior question the pattern's popularity tends to skip: do you need retrieval at all? It was rhetorical when the pattern was named. With 2026 context windows above a million tokens and the cached-prefix pricing of section 5.1, it no longer is. A small, stable corpus can be placed whole in the cached prefix of every request: no index, no chunker, no recall to measure, and the per-request cost of the corpus discounted to a fraction of its first send.

The crossover has four terms, and each is measurable. **Corpus size against the window:** an employee handbook fits, a customer's contract archive does not. **Change rate against cache invalidation:** any edit to the corpus rewrites the prefix and re-pays the uncached rate fleet-wide, so a corpus that changes daily erodes the discount that made stuffing viable. **Per-query cost of resending, even discounted, against the cost of maintaining an index.** And **the grounding requirement:** retrieval hands you a small set of identified passages to cite and check the answer against, which sections 6.8 and 6.9 depend on; a model answering from half a million stuffed tokens offers no equivalent handle, and provenance must be reconstructed after the fact.

docassist fails the stuffing test on all four terms: the corpus is multi-tenant, larger than any window, changes whenever a customer uploads, and exists to produce checkable citations. That is why the rest of this chapter builds retrieval. When a project genuinely passes on all four, build the stuffed-prefix version and write down which term would have to flip before an index earns its keep.

6.2 Document Extraction and Layout Parsing

This is where most retrieval quality is won or lost, and it happens before a single embedding is computed. **Everything downstream inherits the extraction, and no amount of retrieval tuning recovers information that was destroyed at page one.**

A basic PDF loader returns a flat character stream per page. Four things are lost, and each has a specific downstream consequence.

What is destroyed	How it presents later
Table structure	Rows and columns interleave into prose. A rate table becomes a sequence of numbers with no association to their labels, and the model confidently pairs the wrong ones.
Multi-column layout	The reading order interleaves the columns, so sentences from unrelated paragraphs merge. This is the most damaging and the least visible: the text looks fine at a glance.
Heading hierarchy	Section titles become indistinguishable from body text, so the chunker in 6.5 has nothing to split on and the `section_path` in 6.4 cannot be built.
Headers and footers	Repeated on every page, they inflate the corpus with near-duplicate text that competes with real content in retrieval.

The output of extraction should not be a string. It should be a list of typed blocks that preserve what the layout meant:

```
# src/docassist/retrieval/chunking.py
@dataclass(frozen=True)
class Block:
    """One layout element from the parser, before chunking."""
    kind: str          # "heading" | "paragraph" | "table" | "list" | "caption"
    text: str
    page: int
    level: int = 0     # heading depth, 0 for body
```

That `kind` field is what makes the structural splitter of section 6.5 possible. A splitter given a flat string can only count characters; a splitter given typed blocks can decide that a table is never cut and that a heading opens a new unit.

The tooling that produces those blocks falls into three classes, and the classes are more stable than the tool names, which change year to year. **Rule-based PDF parsers** read the text layer directly: fast, cheap, exact where the layer is clean, helpless on scans and hostile layouts. **Layout-model pipelines** run a detection model over the page image to recover regions, tables and reading order, then attach the text layer to the recovered structure: slower, and the standard choice for structured documents. **Vision-model transcription** sends the page image whole to a multimodal model and asks for structured text back: the most capable on the worst inputs, the most expensive, and probabilistic where the other two are exact.

The property that decides between them is not a benchmark score. It is whether the output preserves reading order, tables and heading structure as typed blocks. An extractor that returns beautiful flat prose has already destroyed what section 6.5 needs, however well it transcribed the words.

And measure extraction directly before blaming retrieval for anything: sample twenty pages, put the extracted blocks beside the rendered page, and look. It is the least glamorous evaluation in this book and the most frequently skipped, and a garbled table found this way explains a class of failures that no retrieval tuning will ever fix.

Where vision fits, per section 3.7

The right default is **text extraction first, with vision as a per-page fallback**, not a per-document choice. Decide by measuring what extraction returned:

```
def needs_vision(page_blocks: list[Block], page_area_covered: float) -> bool:
    """Route per PAGE, not per document. A contract is often 200 pages of
    clean digital text with three scanned signature pages in the middle."""
    text_chars = sum(len(b.text) for b in page_blocks)
    if text_chars < 200:                      # extraction returned nothing
        return True
    if page_area_covered < 0.15:              # text found, but sparse
        return True
    return any(b.kind == "table" and _looks_garbled(b.text) for b in page_blocks)
```

Sending every page as an image is simpler to build, costs roughly an order of magnitude more, and *measures worse* on text-heavy documents, because extracted text is exact while vision transcription is probabilistic.

> THE VISUAL-EMBEDDING ALTERNATIVE
>
> By 2026 there is a considered alternative that removes extraction entirely: embed the page images themselves and retrieve pages, with late-interaction scoring between query tokens and page patches. On scanned or table-heavy corpora it retrieves well precisely where extraction is weakest, and there is no parse to get wrong.
>
> The cost is that everything this chapter builds on text spans stops working. Citations point at a page rather than a clause, the entailment check of section 6.9 has no span to check a claim against, and the model must read the answer off an image at generation time. Treat it as a deliberate trade for corpora where extraction demonstrably fails, measured the same way as any other retrieval choice, not as a way to skip the unglamorous work.

6.3 Local Vector DB Setup with Docker Compose

The compose service exists from section 1.3. What matters here is the collection configuration, because two of its settings are decisions rather than defaults.

```
# The distance metric must match how your embeddings were trained.
# Getting this wrong does not error. It degrades retrieval silently, which
# is the section 1.2 failure mode arriving through the index.
COLLECTION = {
    "vectors": {"size": 1536, "distance": "Cosine"},
    # Payload indexes on the fields you FILTER by. Without them, a tenant
    # filter degenerates into a post-filter scan and your p95 collapses as
    # the corpus grows.
    "payload_indexes": ["tenant_id", "doc_id", "is_current_version"],
}
```

The `size` is a decision too, not a default: it must equal the output dimensionality of the embedding model you chose, and section 2.9 gave the trade behind choosing it, dimensionality against storage, latency and recall. The 1536 here records that choice; it does not make it.

The payload index point is worth dwelling on. A vector search with a metadata filter can be executed two ways: filter first and search the survivors, or search first and discard non-matches. The second is what you get without an index, and on a multi-tenant corpus it means retrieving twenty results, discarding nineteen belonging to other tenants, and returning one. Correct, and slow, and it degrades as your customer count grows rather than as your document count grows, which makes it a surprise.

> THE INDEX BUILD ID BELONGS IN YOUR CONFIG
>
> Section 1.2 established that a vector index is a materialisation of a specific embedding model at a specific version, and should be handled like a schema change. That means the collection name carries the build: `chunks_emb_v3_1536`, not `chunks`. A collection named for its contents can be dual-written and cut over. A collection named `chunks` can only be rebuilt in place, with a window where retrieval is half-migrated.

6.4 Document Loaders and the Ingestion Contract

Here is the decision that separates a system that can cite from one that cannot, and it is made in a dataclass.

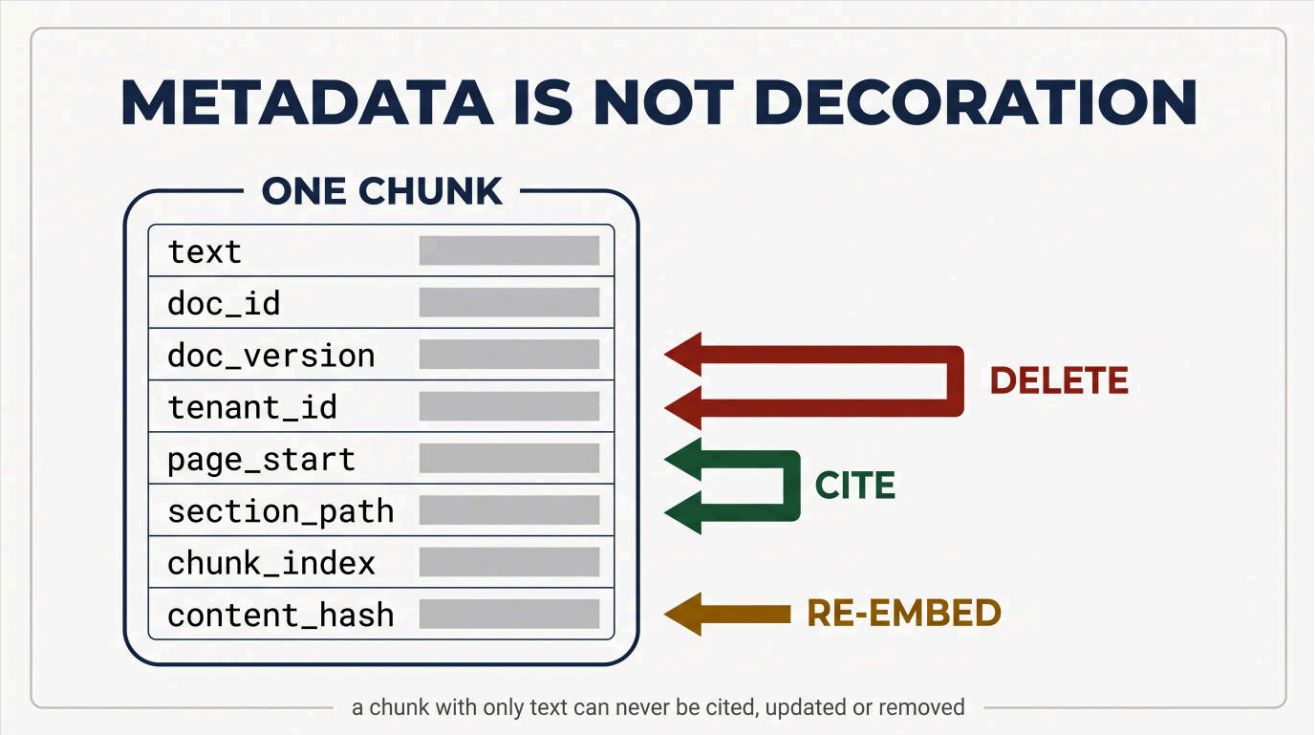

Figure 6.1 Each field exists because something downstream is impossible without it.

```
@dataclass(frozen=True)
class Chunk:
    text: str

    # --- identity and lifecycle ----------------------------------------
    doc_id: str          # DELETE: purge every version by this
    doc_version: int     # DELETE: supersede the previous version
    tenant_id: str       # ISOLATION: enforced at query time, never optional
    content_hash: str    # RE-EMBED: idempotent upsert, skip unchanged work

    # --- citation ------------------------------------------------------
    page_start: int      # CITE: a human must be able to check the claim
    page_end: int
    section_path: str    # CITE, and retrieval: "4. Payment > 4.2 Late fees"

    # --- ordering and expansion ----------------------------------------
    chunk_index: int
    parent_id: str | None = None   # small-to-big expansion (section 7.4)
```

Three of these are worth defending individually, because they are the ones people leave out.

Why the embedded text is not the displayed text

```
def embedding_text(self) -> str:
    """What gets EMBEDDED is not what gets SHOWN.

    Prefixing the section path measurably improves retrieval on structured
    documents, because it restores context the chunk lost when it was cut out
    of the whole.
    """
    return f"{self.section_path}\n\n{self.text}" if self.section_path else self.text
```

A chunk reading "shall not exceed 30 days" is nearly unretrievable in isolation: it contains no noun that a question would match on. The same chunk embedded as "Contract ABC > 4. Payment > 4.2 Late fees: shall not exceed 30 days" matches a question about late payment terms directly. The user still sees only the clause; the index sees the clause in its place.

Why the content hash excludes position

```
def content_hash(text: str, section_path: str) -> str:
    """Hash what determines the EMBEDDING, not the whole chunk.

    Including chunk_index would make every hash change when a paragraph is
    inserted earlier in the document, which defeats the purpose: you would
    re-embed the entire document for a one-line edit.
    """
    return hashlib.sha256(f"{section_path}\x00{text}".encode()).hexdigest()[:16]
```

This is a small decision with a large operational consequence. A four-hundred-page contract receives an amendment adding one clause on page three. With position in the hash, every subsequent chunk shifts index and every hash changes, so you re-embed four hundred chunks. Without it, you re-embed one. Multiply by a customer base that amends documents regularly and the difference is a line item.

Why the parent id is there before you need it

`parent_id` is unused until section 7.4 builds small-to-big retrieval. It is populated now because backfilling it means reprocessing every document you have ingested. Fields that are cheap to write and expensive to add later belong in the first version of the contract.

> THE TEST FOR AN INGESTION CONTRACT
>
> Ask three questions of a stored chunk. *Can a user check this claim?* needs page and section. *Can I re-ingest this document without paying twice?* needs the content hash. *Can I delete this customer's document and have it be gone?* needs doc id, version and tenant. If any answer is no, you are storing text and calling it a chunk.

6.5 Chunking: Fixed, Recursive and Structural

Chunk size is discussed as a tuning parameter and behaves like a structural choice, because it determines what a *retrievable unit is*. You are not adjusting a number; you are deciding what the smallest addressable piece of your corpus will be, forever, until you reprocess everything.

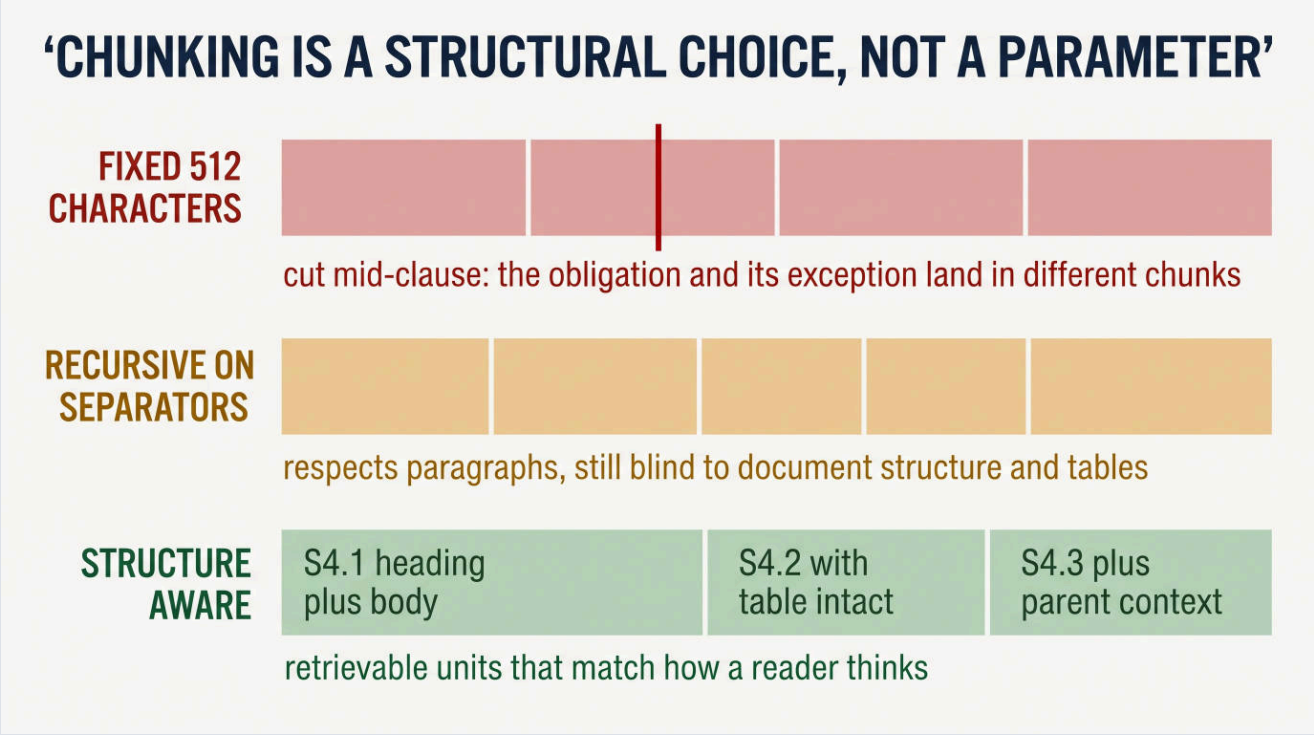

Figure 6.2 The same document under three strategies. Only the third produces units that correspond to how a reader thinks about the text.

GOOD

Fixed size

```
def split_fixed(blocks: list[Block], size: int = 512) -> list[str]:
    joined = "\n".join(b.text for b in blocks)
    return [joined[i : i + size] for i in range(0, len(joined), size)]
```

A fixed window has no relationship to the document's structure. It routinely separates an obligation from its exception, a table from its caption, and a definition from the term it defines. The overlap parameter people add is a partial mitigation that mostly adds cost, because it duplicates content into the index without restoring the semantic unit.

Note what the function returns: **strings**. Not chunks. There is no page, no section, no hash, and therefore no citation, no idempotent upsert and no deletion path. The type signature is the tell.

BETTER

Recursive on separators

Split on paragraph breaks first, then sentences, then characters, recursing only when a unit exceeds the size limit. This respects paragraph boundaries, which is a real improvement, and it remains blind to document structure: it cannot tell a heading from a sentence or a table from a paragraph, because it never saw the block types from section 6.2.

BEST

Structure aware

```
def split_structural(blocks, *, doc_id, doc_version, tenant_id,
                     max_tokens=700, count, overlap_tokens=60) -> list[Chunk]:
    """Split on document STRUCTURE first, size second.

    Two invariants a size-driven splitter cannot provide:
      1. A table is never split. It becomes its own chunk or it stays whole.
      2. A chunk never begins mid-section: the heading path travels with it.

    A chunk that ends mid-sentence is a defect, not a tuning parameter.
    """
    out: list[Chunk] = []
    heading: list[str] = []        # the section path so far, deepest last
    current: list[Block] = []

    def flush(buf: list[Block]) -> None:
        ...   # emit buf as one Chunk carrying " > ".join(heading) as its section_path

    for b in blocks:
        if b.kind == "heading":
            flush(current); current = []
            heading = heading[: max(0, b.level - 1)] + [b.text.strip()]
            continue

        if b.kind == "table":
            flush(current); current = []
            flush([b])                      # tables stand alone, always
            continue

        if count("\n".join(x.text for x in current + [b])) > max_tokens:
            flush(current)
            current = _tail_overlap(current, overlap_tokens, count)

        current.append(b)

    flush(current)
    return out
```

Run it on a contract fragment and the invariants are visible in the output:

```
[acme-msa:2:0] p.3  4. Payment                 'Payment shall be made within 30 days...'
[acme-msa:2:1] p.3  4. Payment > 4.2 Late fees  'Interest accrues at 2% per month...'
[acme-msa:2:2] p.4  4. Payment > 4.2 Late fees  '| Tier | Rate |\n| A | 2% |\n| B | 3% |'
[acme-msa:2:3] p.4  5. Termination              'Either party may terminate on 60 days...'

# same document, fixed splitter: 3 opaque strings, no page, no section, no hash
```

The table survived whole and kept the heading path of the section it belongs to, so a question about late fee rates retrieves a table that still has its labels attached.

The overlap decision

```
def _tail_overlap(buf: list[Block], tokens: int, count) -> list[Block]:
    """Carry one sentence of context, not a fixed character window.

    Overlap exists so a chunk boundary does not orphan a pronoun or a
    subordinate clause. Carrying whole BLOCKS keeps the boundary clean.
    """
```

Character-window overlap duplicates arbitrary fragments and produces chunks that begin mid-word. Block-level overlap carries whole sentences, costs less duplication, and keeps every chunk independently readable, which matters because a user will read the chunk you cite.

6.6 Embedding, Upserting and Idempotent Re-Ingestion

Upload the same contract twice into the tutorial pipeline and the corpus now contains two copies. Both are retrievable, both are citable, and your top four results are two copies of two chunks. **Effective retrieval breadth has halved and nothing reported an error.**

```
async def upsert(self, chunks: list[Chunk]) -> UpsertReport:
    """Idempotent by content hash."""
    doc_id, version = chunks[0].doc_id, chunks[0].doc_version
    existing = await self.index.hashes_for(doc_id=doc_id, doc_version=version)

    incoming = {c.content_hash: c for c in chunks}
    new = [c for h, c in incoming.items() if h not in existing]
    stale = existing - set(incoming)

    if new:
        vectors = await self.embed_batch([c.embedding_text() for c in new])
        await self.index.upsert([(c.id, v, _payload(c)) for c, v in zip(new, vectors)])

    deleted = 0
    if stale:
        deleted = await self.index.delete_ids(
            f"{doc_id}:{version}:{h}" for h in stale)
    return UpsertReport(embedded=len(new), skipped=len(chunks) - len(new),
                        deleted=deleted)
```

Three behaviours fall out. Re-ingesting an **unchanged** document is a no-op costing one index read. Re-ingesting a **changed** document embeds only what changed, which on a typical amendment is three chunks of four hundred. And chunks that vanished from the source are **removed**, so a deleted clause stops being retrievable rather than lingering as a ghost that contradicts the current text.

That last one is the subtle failure of append-only ingestion: the index accumulates every version of every sentence a document has ever contained, and retrieval cheerfully returns the 2023 payment terms alongside the 2026 ones with no way for the model to know which is current.

6.7 Vector Stores as Retrievers

The retriever is where the constraints of Chapter 2 become parameters, and where the most common quality mistake in this field is made.

```
async def retrieve(self, question: str, scope: QueryScope, *, k: int = 20):
    """Retrieve BROADLY here. Narrow in the reranker (7.7).

    The instinct is to set k low because the prompt budget is small. That
    conflates two different decisions: how many CANDIDATES to consider, and
    how many to SHOW the model. Section 2.10 constrains the second. The first
    should be generous, because a chunk that is never retrieved can never be
    reranked into first place.
    """
    return await self.index.search(
        await self.embed(question),
        limit=k,
        filter=assert_scoped(scope.as_filter()),
    )
```

Retrieve twenty, show four. Setting k to four at the retriever means the reranker has nothing to work with and the abstain path in section 6.9 cannot distinguish "no good evidence exists" from "we only looked at four things".

> THE TRAP THIS SETS FOR YOUR METRICS
>
> Retrieval metrics and answer quality can move in opposite directions. Raising k from four to twenty raises recall almost by definition and, if you feed all twenty to the model, *reduces* answer accuracy through the dilution of section 2.10.
>
> The resolution is that k at the retriever and k at the prompt are different numbers with different owners. Section 7.10 makes this the central point about measuring retrieval, because a team that optimises recall@k without an end-to-end metric will build a system that scores well and answers badly.

6.8 Assembling the Prompt and Citing Sources

Assembly is where Chapter 2's budget and Chapter 5's registry meet the evidence.

```
def render_evidence(chunks: list[Chunk]) -> str:
    """Render with STABLE, SHORT ids the model can cite and code can resolve.

    E1, E2, E3 rather than document filenames or UUIDs: short ids cost fewer
    tokens, are far less likely to be mistyped by the model, and are trivially
    validated against the retrieved set (5.3).
    """
    return "\n\n".join(
        f'<source id="E{i}" doc="{c.doc_id}" page="{c.page_start}">\n'
        f"{c.section_path}\n{c.text}\n</source>"
        for i, c in enumerate(chunks, 1)
    )


assembled = budget.assemble({
    "system": spec.system,
    "evidence": render_evidence(top),
    "history": compacted_history,
    "query": question,
}, count=count_tokens)

if assembled.had_pressure:
    # Section 2.10: overflow is an EVENT, not a silence. If the evidence
    # region was truncated, the model never saw part of what you retrieved,
    # and the trace must say so or the failure is unattributable.
    tracer.record_budget_events(assembled.events)
```

Two ordering decisions follow from section 2.7's positional degradation, and both are engineering choices rather than defaults.

Evidence goes in reranked order, best first. The model attends most reliably to the start and end of the context, so the highest-scoring passage should not be buried in the middle of twenty.

The instruction that governs the output goes last, immediately before the answer is requested. Repeating the citation requirement there costs a dozen tokens and measurably improves compliance, because it sits in the position the model attends to most.

Citations are resolved, not trusted

Section 5.3 established that a citation can parse and still be a fabrication. The resolution step belongs here, in the assembly and answer path, because this is where the mapping from `E1` to a real chunk exists:

```
result = await runner.run("doc_qa", variables, schema=CitedAnswer)
verify_citations(result, evidence_count=len(top))     # E9 with 4 sources fails
rendered = [(cid, top[int(cid[1:]) - 1]) for cid in result.citations]
```

What the user sees is not `[E1]` but the document name and page it resolves to, which is the whole point: **a citation the user cannot follow is decoration.**

6.9 Grounding Verification and the Abstain Path

The model was asked to cite and complied syntactically. That is not the same as being grounded, and this section adds the checks that come after citation resolution.

Three checks, in increasing cost

```
class RelevanceFloor:
    """Refuse BEFORE generating when the evidence is weak.

    The cheapest of the three: one comparison, and it prevents a whole class
    of confident fabrication, because a model given only irrelevant passages
    will still write something.

    The floor is a MEASURED number from your eval set, not a guess. Set it
    where abstaining starts costing more answers than it saves wrong ones.
    """

    def passes(self, evidence: Sequence[Evidence]) -> bool:
        return bool(evidence) and evidence[0].score >= self.min_top_score
```

The second check is citation resolution from section 5.3: does every cited id exist in the retrieved set. The third is entailment, and one detail in it matters more than the rest:

```
class GroundingChecker:
    """Claim-level entailment against the CITED span.

    Deliberately checks each claim against the span it cited, NOT against the
    whole evidence set. Checking against everything lets a model cite E1 while
    the support actually lives in E3, which is a citation that is technically
    resolvable and still misleading to a reader who follows it.
    """

    async def verify(self, claims, evidence) -> GroundingReport:
        by_id = {e.id: e for e in evidence}
        for claim, cited_id in claims:
            span = by_id.get(cited_id)
            if span is None or not await self._entails(claim, span.text):
                unsupported.append(claim)
```

```
async def test_a_claim_cited_to_the_wrong_span_is_unsupported():
    """Checking against the WHOLE evidence set would pass this. Checking
    against the cited span, which is what a reader follows, does not."""
    ev = [Evidence("E1", "Payment is due in 30 days.", 0.9),
          Evidence("E2", "Termination requires 60 days notice.", 0.8)]
    report = await GroundingChecker(entails).verify(
        [("60 days notice", "E1")], ev)      # correct fact, wrong citation
    assert report.fully_grounded is False
```

That case is a true statement, drawn from the corpus, attached to the wrong source. A reader who clicks the citation finds payment terms and concludes the system is unreliable, which is the correct conclusion.

The argument for abstention, and how to win it

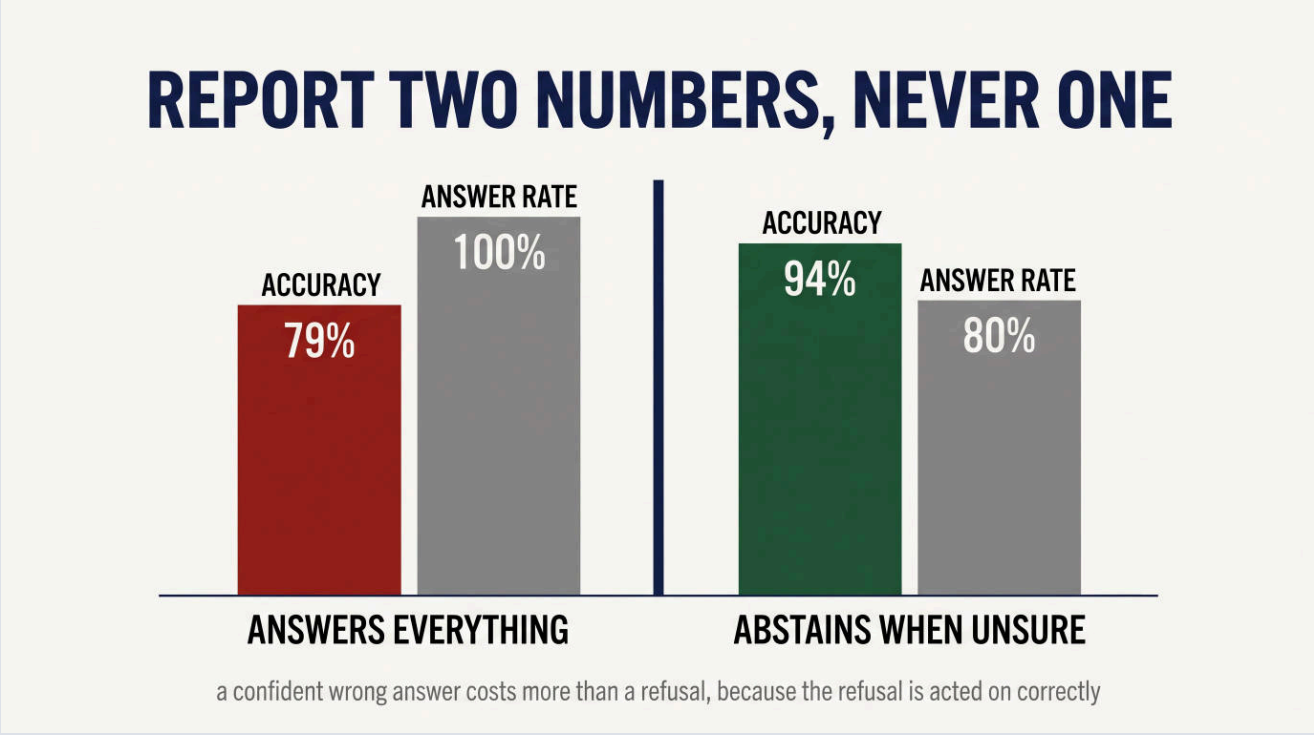

Figure 6.3 The two-number form is what makes the trade visible. Reporting accuracy alone rewards a system that guesses.

Every product team resists the abstain path, because "I could not find this in the documents" feels like failure. It is the opposite, and the argument that wins the meeting is arithmetic rather than principle.

Report two numbers instead of one: accuracy *on answered questions*, and answer rate. A system at 94 percent accuracy on 80 percent of questions is usually worth more than one at 79 percent on all of them, and only the two-number form makes that visible.

```
def accuracy_on_answered(answers, correct) -> float:
    """Report TWO numbers, never one. Reporting accuracy alone rewards a
    system that guesses."""
    answered = [(a, c) for a, c in zip(answers, correct) if a.answered]
    return sum(c for _, c in answered) / len(answered) if answered else 0.0
```

The underlying asymmetry: in professional document work, **a confident wrong answer costs far more than a**

refusal, because the refusal is correctly acted on and the wrong answer is not. A contracts manager told "the notice period is 30 days" when it is 60 does not double-check; that is what they asked the system for.

> ABSTENTION MUST BE VISIBLY DISTINCT
>
> Section 3.6 made the same point about degraded responses and it applies here. An abstention that reads like a hedged answer trains users to skim past it. Return a distinct state, render it differently, and say what was searched: "No passage in the three documents in scope addresses the notice period." That sentence is actionable. "I could not find a definitive answer" is not.

6.10 Multi-Tenancy, Versioning and Deletion

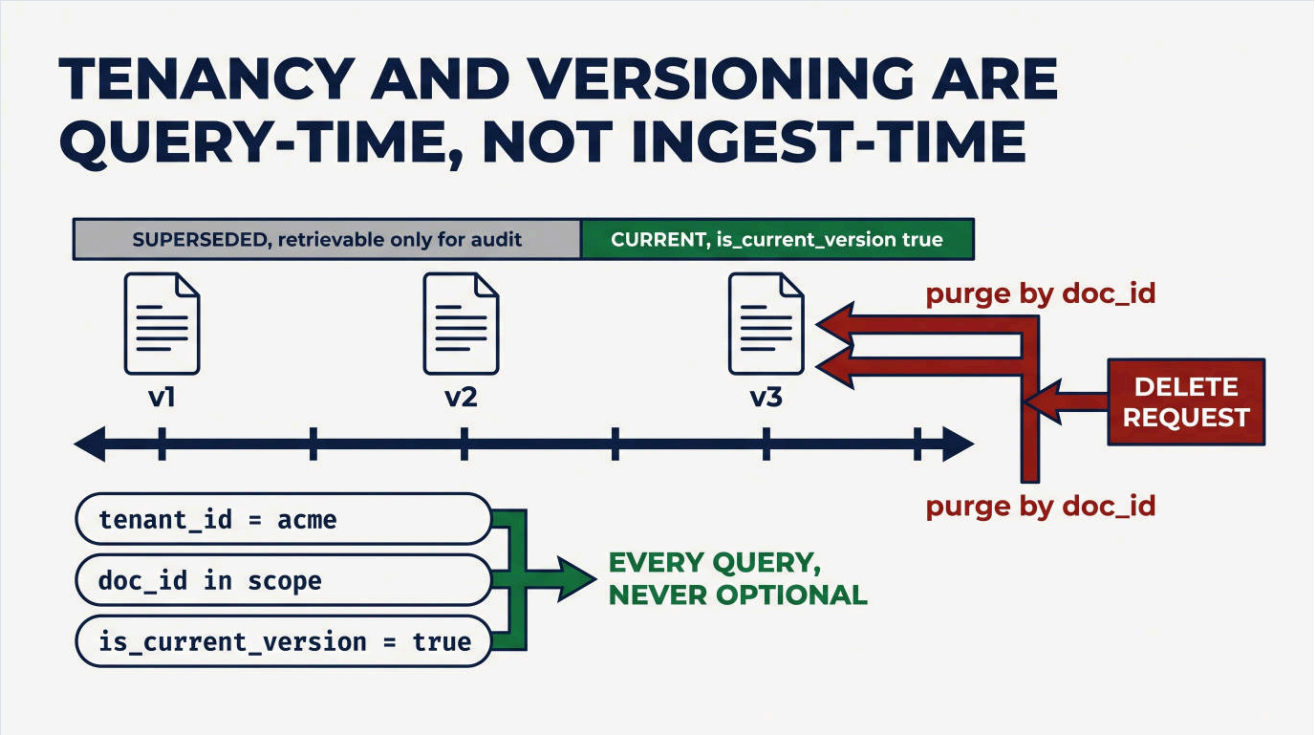

Figure 6.4 Storing the tenant on the chunk is necessary and not sufficient. Something must apply it on every read.

Three capabilities, all trivial if designed for and close to impossible to retrofit.

Isolation is a query-time property

```
@dataclass(frozen=True)
class QueryScope:
    tenant_id: str
    doc_ids: tuple[str, ...] | None = None
    current_only: bool = True


def assert_scoped(filter_: dict) -> dict:
    """Called by the retriever before EVERY search. Cheap, and it has caught
    a real class of bug: a code path that built its own filter dict and
    forgot the tenant."""
    if not filter_.get("tenant_id"):
        raise PermissionError("retrieval attempted without a tenant filter")
    return filter_
```

Making the scope a *required argument* rather than an optional keyword, and asserting on it at the boundary, is the difference between a control and a convention. The failure this prevents is not hypothetical: a background job, a debug endpoint or an admin tool builds its own filter, omits the tenant, and one customer's contract appears in another's search results.

The same pattern extends below the tenant. Per-user permissions within a tenant are a per-document ACL field on the chunk payload, asserted at query time exactly as the tenant is. One rule is absolute: the values in that filter come from the authenticated identity on the request, never from anything the model produced.

Versioning is supersession, not replacement

```
async def promote(self, doc_id: str, version: int) -> None:
    """Cut over AFTER the new version is fully indexed.

    Doing this first would leave a window where the new version is partially
    present and the old one is already unreachable, which shows up to users
    as a document that briefly forgot half its content.
    """
    await self.index.set_current_version(doc_id, version)
```

Superseded versions stay retrievable under an explicit scope, because "what did this contract say in March" is a real question in professional document work, and because deleting history to save storage is a decision you make once and regret during an audit.

Deletion must actually delete

When a customer deletes a document, the vectors must go. In the tutorial pipeline they do not, because nothing links a vector back to a document. This is a compliance problem long before it is an engineering one, and it is the clearest example of a Chapter 6 decision determining whether a Chapter 15 requirement is satisfiable at all.

6.11 Three Architectures Compared

GOOD

The tutorial pipeline

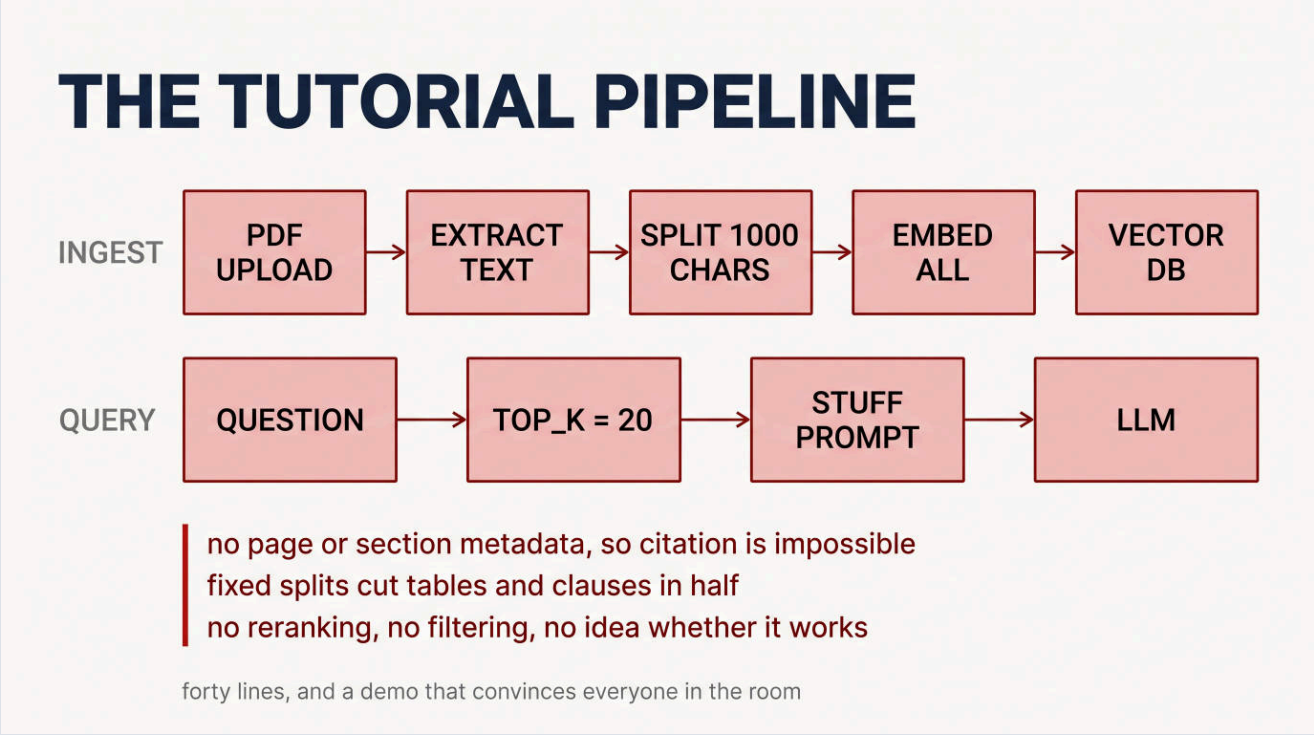

Figure 6.5 Eight components, forty lines, and a demo that convinces everyone in the room.

It is correct for one purpose: proving that retrieval helps at all, on documents you control, in under a day. That is a real and valuable use. Everything fails the moment documents arrive that you did not choose.

BETTER

Structured ingestion, precision retrieval

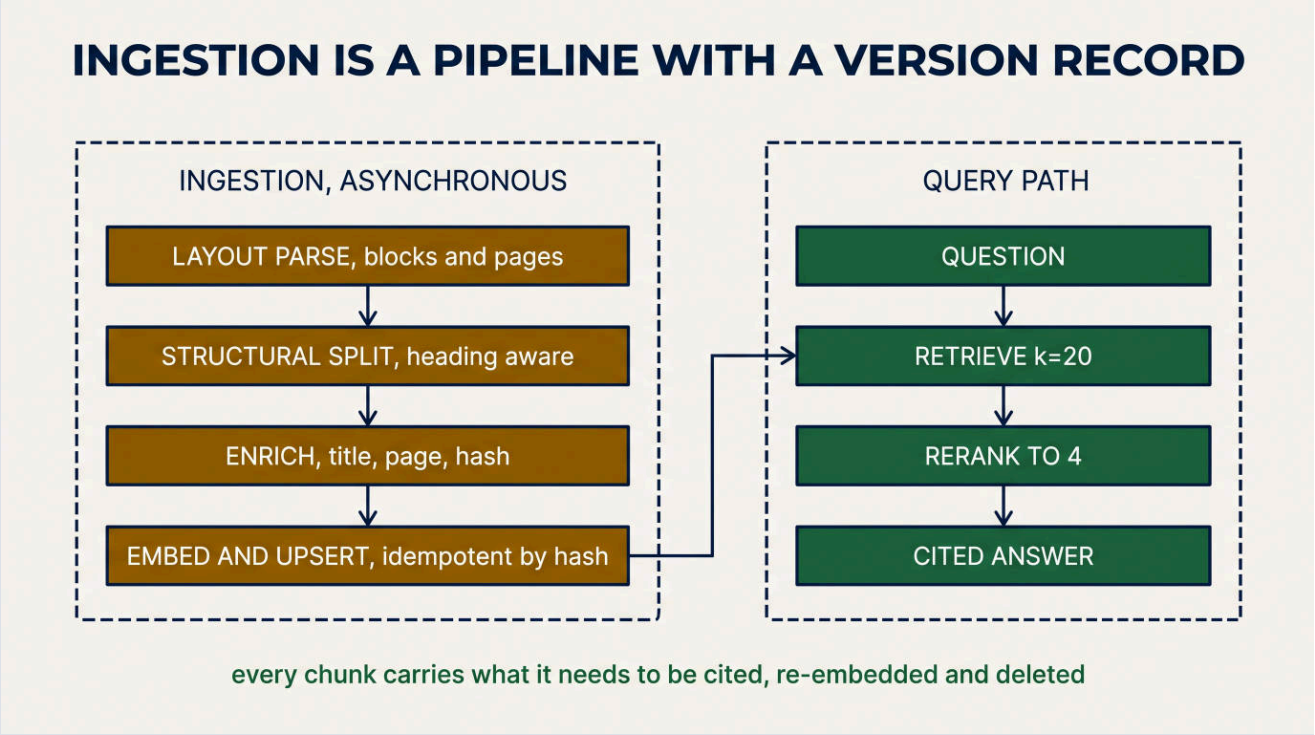

Figure 6.6 Ingestion becomes a versioned pipeline; retrieval becomes precision-first and citable.

What it still does not solve. Retrieval is purely semantic, so exact identifiers, part numbers, clause references and rare proper nouns retrieve poorly, for the reason section 2.9 gave: embeddings are topical. A question needing two documents joined is not served, because retrieval is a single flat search. And nothing checks whether the answer is actually supported.

BEST

Hybrid, reranked, grounded, willing to abstain

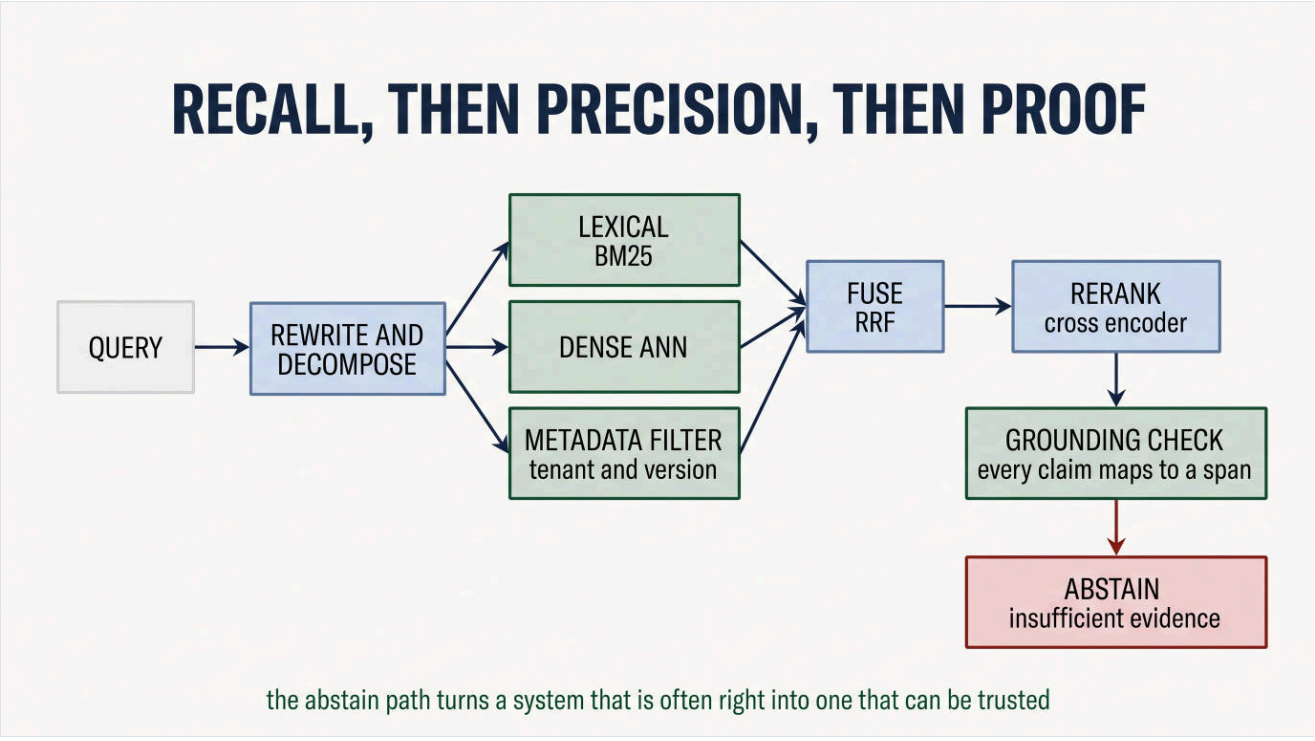

Figure 6.7 Broad recall from complementary signals, aggressive narrowing, then a check that can refuse.

Chapter 7 develops each component. The shape is worth stating on its own, because it is the architecture the rest of Part II defends: **broad recall through complementary retrievers, aggressive narrowing through a reranker, and a verification stage that can say no.**

Axis	Good: tutorial	Better: structured	Best: hybrid and grounded
Citations	Impossible	Page and section accurate	Verified against the cited span
Exact term retrieval	Poor	Poor	Strong via the lexical leg
Re-ingestion	Duplicates silently	Idempotent, versioned	Idempotent, versioned
Deletion and tenancy	Absent	Enforced at query time	Enforced at query time
Wrong answer rate	High and invisible	Lower, still unmeasured	Bounded by grounding and floor
Build cost	One day	Two to three weeks	Six to ten weeks

PROMOTION TRIGGERS

Good to better, before any document you did not personally choose enters the system, which in practice means before the first external user. Structured ingestion is not an optimisation; it is the difference between a system that can cite and one that cannot, and retrofitting it means reprocessing your entire corpus.

Better to best, when your failure log shows exact-term misses (part numbers, clause references, names), when questions require joining two sources, or when a wrong answer carries professional consequence. **Build the lexical leg first**: it is a day of work and typically the largest single retrieval gain available. Build grounding verification last, because it is the most expensive and depends on everything else being solid.

What Chapter Six Establishes

What exists now	What later chapters put through it
Typed blocks from layout parsing	Chunking strategy comparison and measurement *(7.3)*
Chunk contract with parent ids	Small-to-big retrieval and contextual prefixing *(7.4)*
Idempotent upsert and version supersession	Bulk reingestion on a separate queue class *(8.9)*
Retrieve-broad, show-narrow separation	Hybrid fusion and cross-encoder reranking *(7.5 to 7.7)*
Citation resolution against the retrieved set	Claim-level entailment and faithfulness metrics *(7.10)*
Relevance floor and abstain state	Failure diagnosis by stage *(7.2)*, designed degradation *(15.3)*
Query scope enforced at the boundary	PII handling and data retention *(15.4)*

Chapter checklist

Can you...	Section
State what RAG is solving in terms of the Chapter 2 constraints	6.1
Name four things a flat text extractor destroys, and their downstream effects	6.2
Explain why a payload index turns a filter into a pre-filter	6.3
Justify every field on the chunk contract by what it makes possible	6.4
Say why the embedded text differs from the displayed text	6.4
Explain why the content hash must exclude chunk position	6.4, 6.6
Give the two invariants a structural splitter provides and a size-driven one cannot	6.5
Say why k at the retriever and k at the prompt are different numbers	6.7
Explain why entailment is checked against the cited span, not the whole set	6.9
Win the abstention argument with two numbers	6.9
Say why tenancy is a query-time property enforced by an assertion	6.10

What Chapter 7 builds

Chapter 6 built a system that can cite and can refuse. Chapter 7 is the highest-leverage chapter in the book, and it starts by refusing to optimise anything until the failures have been diagnosed.

A RAG answer can fail at five distinct stages, and the fix at each is different and non-transferable. Section 7.2 gives the procedure for labelling fifty failures by stage, which takes an afternoon and is almost always surprising: the majority of failures occur *before the model is ever called*, and are therefore invisible to anyone who only reads the answers. Then chunking strategies compared on measurement rather than assertion, hybrid retrieval and why lexical search refuses to

die, cross-encoder reranking, query rewriting, and the retrieval eval set that everything else in Part II depends on.